Integration of Informal and Formal Methods for the Reverse Engineering of C Programs*

Gerald C. Gannod† and Betty H. C. Cheng‡

Department of Computer Science
Michigan State University
East Lansing, Michigan 48824
tel: (517) 355-8344 fax: (517) 432-1061
{gannod,chengb}@cps.msu.edu

Abstract

Reverse engineering of program code is the process of constructing a higher level abstraction of an implementation in order to facilitate the understanding of a system that may be in a "legacy" or "geriatric" state. Changing architectures and improvements in programming methods, including formal methods in software development and object-oriented programming, have prompted a need to reverse engineer and re-engineer program code. This paper describes the an integrated approach that incorporates the use of semi-formal analysis and formal program semantics to reverse engineer C programs.

1 Introduction

Software maintenance has long been a problem faced by software professionals, where the average age of software is between 10 to 15 years old [1]. With the development of new architectures and improvements in programming methods and languages, including formal methods in software development and object-oriented programming, there is a strong motivation to reverse engineer and re-engineer existing program code in order to preserve functionality, while exploiting the latest technology. Formal methods in software development provide many benefits in the forward engineering aspect of software development [2]. One of the advantages of using formal methods in

*This work is supported in part by the National Science Foundation grants CCR-9407318, CCR-9209873, and CDA-9312389 and NASA Grant NGT-70376

†This author is supported in part by a NASA Graduate Student Researchers Program Fellowship

‡Please address all correspondences to this author.

software development is that the formal notations are precise, verifiable, and facilitate automated processing [3]. *Reverse Engineering* is the process of constructing high level representations from lower level instantiations of an existing system. One method for introducing formal methods, and therefore taking advantage of the benefits of formal methods, is through the reverse engineering of existing program code into formal specifications [4, 5, 6].

This paper suggests an approach for integrating the use of informal methods, such as structured analysis, with formal techniques in order to reverse engineer imperative programs written in the C programming language. The formal approach is based on the formal semantics of the *strongest postcondition* predicate transformer *sp* [7], and the partial correctness model of program semantics introduced by Hoare [8]. The objective of this integrated approach is to take advantage of the benefits of graphical notations while providing a rigorous underlying formalism. The integrated approach is applied to actual source code taken from an existing NASA application involving unmanned flight systems. Previously, we investigated the use of the *weakest precondition* predicate transformer *wp* as the underlying formal model for constructing formal specifications from program code [4, 9]. More recently, we described the use of *sp* as a formal basis to reverse engineering programs written in Dijkstra's guarded command language [10, 11].

The remainder of this paper is organized as follows. Section 2 provides background material for software maintenance and formal methods. The semantics of the C programming language using the strongest postcondition predicate transformer is described in Section 3. The issues related to integrating informal and formal methods for reverse engineering are discussed in Section 4, and this approach is applied to a NASA ground-based system for controlling unmanned spacecraft in Section 5. Related work is described in Section 6. Finally, Section 7 draws conclusions, and suggests future investigations.

2 Background

This section provides background information for software maintenance and formal methods for software development. Included in this discussion is the formal model of program semantics used throughout the paper.

2.1 Software Maintenance

Figure 1 contains a graphical depiction of a process model for reverse and re-engineering [12, 13]. The process model appears in the form of two sectioned triangles, where each section in the triangles represents a different level of abstraction. The higher levels in the model are *concepts* and *requirements*. The lower levels include *designs* and *implementations*. The relative size of each of the sections is intended to represent the amount of information known about a system at a given level of abstraction. Entry into this re-engineering process model begins with system A, where *Abstraction* (or reverse engineering) is performed to an appropriate level of detail. The next step is *Alteration*, where the system is constituted into a new form at a different level of abstraction). Finally, *Refinement* of the new form into an implementation can be performed to create system B.

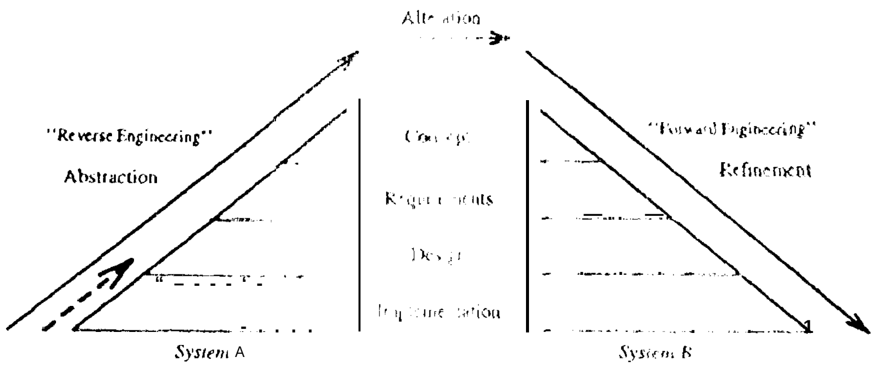


Figure 1: Reverse Engineering Process Model

This paper describes an approach to reverse engineering that is applicable to the *implementation* and *design* levels. In Figure 1, the context for this paper is represented by the dashed arrow.

That is, we address the construction of formal low-level or "*as-built*" design specifications. The motivation for operating in such an implementation-bound level of abstraction is that it provides a means of traceability between the program source code and the formal specifications constructed using the techniques described in this paper. This traceability is necessary in order to facilitate technology transfer of formal methods. That is, currently existing development teams must be able to understand the relationship between the source code and the specifications.

2.2 Formal Methods

Although the waterfall development life cycle provides a structured process for developing software, the design methodologies that support the lifecycle (i.e. Structured Analysis and Design [14]) make use of informal techniques, thus increasing the potential for introducing ambiguity, inconsistency, and incompleteness in designs and implementations. In contrast, formal methods used in software development are rigorous techniques for specifying, developing, and verifying computer software [2]. A formal method consists of a well-defined specification language with a set of well-defined inference rules that can be used to reason about a specification [2]. A benefit of formal methods is that their notations are well-defined and thus, are amenable to automated processing [3].

2.2.1 Program Semantics

The notation Q { S } R [8] is used to represent a partial correctness model of execution, where, given that a logical condition Q holds, if the execution of program S terminates, then logical condition R will hold. A rearrangement of the braces to produce { Q } S { R }, in contrast, represents a total correctness model of execution. That is, if condition Q holds, then S is guaranteed to terminate with condition R true. The context for our investigations is that we are reverse engineering systems that have desirable properties or functionality that should be preserved or extended. Therefore, the partial correctness model is sufficient for these purposes since the termination properties of these systems are known a priori.

2.2.2 Strongest Postcondition

The *strongest postcondition* $sp(S,Q)$ predicate transformer [7] is defined as the set of all states in which *there exists* a computation of S that terminates with Q true. That is, given that Q holds, execution of S results in $sp(S,Q)$ true, if S terminates. As such, $sp(S,Q)$ assumes partial correctness. The *weakest precondition* predicate transformer $wp(S,R)$ is defined as the set of all states in which the statement S can begin execution and terminate with postcondition R true. Given a Hoare triple $Q\ \{S\}\ R$, we note that wp is a "backward" rule, in that a derivation of a specification begins with R, and produces a predicate $wp(S,R)$. The predicate transformer wp assumes a total correctness model of computation, meaning that given S and R, if the computation of S begins in state $wp(S,R)$, the program S *will* halt with condition R true.

We contrast this model with the sp model, a "forward" derivation rule. That is, given a precondition Q and a program S, sp derives a predicate $sp(S,Q)$. The predicate transformer sp assumes a partial correctness model of computation meaning that if a program starts in state Q, then the execution of S will place the program in state $sp(S,Q)$ if S terminates. Figure 2 gives a graphical depiction of the differences between sp and wp, where the input to the predicate transformer produces the corresponding predicate. Figure 2(a) gives the case where the input, to the predicate transformer is "S" and "R", and the output to the predicate transformer (given by the box and appropriately named "wp") is "wp(S,R)". The sp case (Figure 2(b)) is similar, where the input to the predicate transformer is "S" and "Q", and the output to the transformer is "sp(S,Q)".

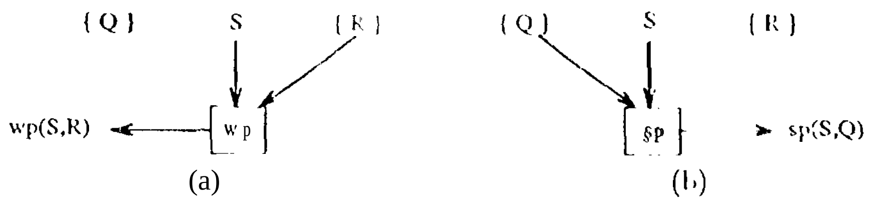


Figure 2: Black box representation and differences between wp and sp: (a) wp (b) sp

The use of these predicate transformers for reverse engineering have different implications. Using *wp* implies that a postcondition R is known. However, with respect to reverse engineering, determining R is the objective, therefore *wp* can only be used as a guideline for performing reverse engineering. The use of *sp* assumes that a precondition Q is known and that a postcondition will be derived through the direct application of *sp*. Therefore, *sp* is more applicable to reverse engineering, and is used as such in this project.

3 Semantics of C Programs

Our previous investigations [10, 11] involved the use of the strongest postcondition predicate transformer as applied to the Dijkstra guarded command language [15]. This section defines the *sp* semantics of the C programming language [16]. Due to space constraints, only a subset of programming language used in the application example is presented. A more complete description of the semantics of the C programming language may be found in [?].

3.1 Assignment

Let u be a variable or an assignable expression and e be an expression. An assignment in the C programming language has the form v $\circledast$ e, where $\circledast$ is an assignment operator (i.e., =, +=, *=). There are two roles that an assignment statement can have. The first is the traditional assignment, of a variable with the value of an expression. The second role is as a side-effect boolean expression.

In order to cope with the dual role of an assignment statement, two functions are defined. First, in order to describe the semantics of the traditional use of assignment, an *evaluation* function $\mathcal{A}: S \rightarrow \mathcal{T}$ is defined, where S is the set of syntactically valid assignment expressions, and $\mathcal{T}$ is the type of the result given by evaluating the expression e. For instance, given an assignment statement "x *= n", the function $\mathcal{A}$ would be evaluated as $\mathcal{A}$(x *= n) = x * n. Table 1 defines the semantics of the function $\mathcal{A}$ on a few selected assignment operators. A more general form of the function $\mathcal{A}$, can be defined as as $\mathcal{A}$(b) [illegible] where S is the set of valid expressions in C., and b *is* an

Operation $\cong$	Evaluation $\mathcal{A}$
$=$	e
$*=$	$v \times e$
$/=$	$\frac{v}{e}$
$+=$	$v + e$
$-=$	$v - e$
$\%=$	$v \bmod e$

Table 1: Evaluation of $\mathcal{A}$ on select C assignment operators

expression. This form is used in the case when the parameter to $\mathcal{A}$ is not an assignment expression. The interpretation is that the evaulation of any expression (that is not an assignment expression) takes the value of the expression. Due to space constraints, we focus primarily on the assignment expressions. Using the definition of $\mathcal{A}$, we can define the strongest postcondition of an assignment in the following manner:

$$sp(\mathbf{x} \cong \mathbf{e}, Q) = (\exists v :: Q^x_v \wedge x = \mathcal{A}(\mathbf{x} \cong \mathbf{e}^x_v)) \tag{1}$$

where Q is the precondition, v is the quantified variable, and '::' indicates that the range of the quantified variable v is not relevant in the current context. This specification states that after the execution of an assignment statement, there exists some value v such that the textual substitution of every free occurrence of x with v in Q keeps Q true, and x takes the value of the evaluation $\mathcal{A}$ on $\mathbf{x} \cong \mathbf{e}^x_v$. This means that after the execution of an assignment statement, the precondition Q must still be true with respect to the value that the variable $\mathbf{x}$ had before the assignment, and that the assignment must be valid.

The second function that is used to define the effects of an assignment statement is the *logical valuation* function $\mathcal{V} : S \rightarrow B$, where B is the Boolean type. The purpose of $\mathcal{V}$ is best motivated by an example. Consider the sequence of code in Figure 3. Informally, the semantics of this code sequence is that if the guard is true, execute S1, otherwise execute S2. However, the guard is peculiar due to the fact that the expression is not a logical one, but rather an assignment expression. The semantics in this case are dependent on the side effect of executing the statement v = e. Using the function $\mathcal{A}$, function $\mathcal{V}$ is defined as:

$$\mathcal{V}(v = e) = \begin{cases} T & \text{if } \mathcal{A}(v = e) \neq 0 \\ F & \text{if } \mathcal{A}(v = e) = 0 \end{cases},$$

where T and F are Boolean constants *true* and *false*, respectively. In general, for some arbitrary expression b, $\mathcal{V}$ is defined as:

$$\mathcal{V}(b) = \begin{cases} T & \text{if } \mathcal{A}(b) \neq 0 \\ F & \text{if } \mathcal{A}(b) = 0\,. \end{cases}$$

Although the side effects of an assignment statement have no effect on the assignment itself, the side effects do impact other operations as was shown in the short example above. In Section 3.2, the use of V will be important for defining the semantics of alternation statements with side-effects.

```
if (v = e) {
   S1
} else {
   S2
}
```

Figure 3: An Assignment statement as a guard

In this paper, discussion has been limited to a small subset of the available assignment operators. The semantics of the shift and bitwise assignment operators can also be defined using the functions $\mathcal{A}$ and $\mathcal{V}$. In addition, the semantics of other expressions can be defined using these functions [?].

3.2 Alternation

The alternation statement for C programs can take two forms:

```
if B {                    if B {
   S                         S_1
}                         } else
                             S_2
```

We refer to these statements as C-IF1 and C-IF2, respectively.

When the guard of an alternation statement has no side effects, the semantics of the alternation statement is as follows:

$$
\begin{aligned}
sp(\texttt{C-IF1}, Q) &\equiv sp(\mathrm{S}, \mathrm{B} \wedge Q) \vee sp(\texttt{skip}, \neg \mathrm{B} \wedge Q) \\
&\equiv sp(\mathrm{S}, \mathrm{B} \wedge Q) \vee (\neg \mathrm{B} \wedge Q)
\end{aligned}
$$

$$
sp(\texttt{C-IF2}, Q) \equiv sp(\mathrm{S}_1, \mathrm{B} \wedge Q) \vee sp(\mathrm{S}_2, \neg \mathrm{B} \wedge Q)
$$

If the restriction of having alternation statements without side- effects in the guards is removed, then the semantics of the alternation statement have a different meaning. Informally, if there is a side-effect in the guard B, then the execution of an alternation is analogous to "executing" B, then running the alternation using the evaluation of B. More formally, let B be a guard of an alternation statement (C-IF1 for instance) such that the evaluation of B causes a side-effect, and let $\mathcal{V}(B)$ represent the truth value of B. Execution of the alternation statement is equivalent to the execution of the following:

```
B;                        B;
if V(B) {                 if V(B) {
   S                         S_1
}                         } else
                             S_2
```

We refer to the alternation statements (the if statement with the replacement of B by $\mathcal{V}(B)$) as C-IF1$_s$ and C-IF2$_s$, respectively. The semantics of C-IF$_s$ are as follows:

$$
\begin{aligned}
sp(\texttt{C-IF1}_s, Q) &\equiv sp(\texttt{C-IF1}, sp(\mathrm{B}, Q)) \\
&\equiv sp(\mathrm{S}_1, \mathcal{V}(\mathrm{B}) \wedge sp(\mathrm{B}, Q)) \vee (\neg \mathcal{V}(\mathrm{B}) \wedge sp(\mathrm{B}, Q))
\end{aligned}
$$

$$
\begin{aligned}
sp(\texttt{C-IF2}_s, Q) &\equiv sp(\texttt{C-IF2}, sp(\mathrm{B}, Q)) \\
&\equiv sp(\mathrm{S}_1, \mathcal{V}(\mathrm{B}) \wedge sp(\mathrm{B}, Q)) \vee sp(\mathrm{S}_2, \neg \mathcal{V}(\mathrm{B}) \wedge sp(\mathrm{B}, Q))
\end{aligned}
$$

3.3 Sequence

Sequences of statements in the C programming language have the form $S_1; \ldots ;S_n$. The appropriate semantics using sp is as follows:

$$sp(S_1;S_2,Q) \quad sp(S_2, sp(S_1,Q)). \tag{2}$$

Since the impact of side-effects are specified by the corresponding sp formalisms for assignment, alternation, and iteration, this characterization of the semantics of sequence is sufficient.

3.4 Iteration

in the C programming language, the iteration construct can take one of the following forms:

```
while (B) {                do {                 for (expr1;expr2;expr3) {
   S;                         S;                   S;
}                          } while (B)          }
```

where B is the guard expression and *expri* are for iteration expressions. This Section describes the strongest postcondition semantics for the while iteration construct of the C programming language. For the do-while and for constructs, appropriate transformations using the while semantics are provided.

3.5 while

When no side effects are present, the while iteration construct has the following semantics:

$$sp(\texttt{while},Q) : \neg B \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i,Q)). \tag{3}$$

Equation 3 states that if the execution of the while statement terminates then the guard B is false and the result of applying the rule $sp(\texttt{C-IF1},Q)$ i times is true. Notationally, $sp(\texttt{C-IF1}^i,Q)$, where

i is the number of iterations, means that sp is recursively applied to the result of $sp(\texttt{C-IF1}, Q)$. For instance, $sp(\texttt{C-IF1}^3, Q)$ has the following derivation:

$$\begin{aligned} sp(\texttt{C-IF1}^3, Q) &\equiv sp(\texttt{C-IF1}, sp(\texttt{C-IF1}^2, Q)) \\ &\equiv sp(\texttt{C-IF1}, sp(\texttt{C-IF1}, sp(\texttt{C-IF1}, Q))). \end{aligned}$$

In the case when the guard of the while statement has a side effect, the semantics are similar to executing the following:

```
B;
while (V(B)) {
    S,
    B;
}
```

where $\mathcal{V}$ is the valuation function described in Section 3.1. The corresponding sp semantics of the **while** statement with side-effects (denoted $\texttt{while}_s$) is

$$sp(\texttt{while}_s, Q) \equiv \neg\mathcal{V}(\texttt{B}) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(\texttt{B}, Q))). \quad (4)$$

where the body of the statement C-IF1 consist of “S; B;”.

3.6 do while

The semantics of the do **while** statement are similar to the **while** statement, where the guarding condition appears after the loop body. Using the **while** construct, do **while** can be written as the following:

```
S;
B;
while (V(B)) {
    S;
    B;
}
```

The corresponding formal specification of the semantics of the do **while** statement is given by Equation 5

$$sp(\texttt{while}_s, sp(S, Q)) \equiv \neg\mathcal{V}(\texttt{B}) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(\texttt{B}, sp(S, Q)))). \quad (5)$$

where the body of the statement C-IF1 consists of "S; B;". This specification states that after the execution of a *do while* statement, the valuation of B is false, and the body of the loop is executed i times, where the initial precondition to the loop is given by $sp(B, sp(S, Q))$.

3.7 for

Recall that the `for` construct in C has the form

```
for (expr1; expr2; expr3) {
    S;
}
```

The semantics of the `for` iteration statement is that the first expression (*expr1*) is executed (evaluated) once, the second expression (*expr2*) is evaluated before each iteration, and the third expression (*expr3*) is evaluated after each iteration. These semantics, using the `while` construct, are represented by the following:

```
expr1;
expr2;
while (V(expr2)) {
    S;
    expr3;
    expr2;
}
```

The resulting formal specification of the semantics of `for` using the *sp* for `while` is

$$sp(\texttt{while}_s, sp(expr1, Q)) \equiv \neg\mathcal{V}(expr2) \wedge (\exists i : 0 \leq i : sp(\text{C-IF1}^i, sp(expr2, Q))). \qquad (6)$$

where the body of the statement C-IF1 consists of "S; *expr3*; *expr2*;". This specification states that after the execution of the `for` loop the logical valuation of *expr2* is false, and the loop body is executed i times where the initial precondition to the loop is given by $sp(expr2, Q)$.

3.8 Function Calls

Functions in the C programming language can serve two basic purposes. A function can be a *pure value function*, where the purpose is to compute some value based on the parameters. Alternatively, a function can be a *procedure*, where the purpose is to perform a number of encapsulated tasks. Our previous investigations [10, 11] describe an approach for defining the semantics of functions that serve a procedural role. Due to space constraints, this discussion is not repeated here.

Table 2 contains a taxonomy of functions based on the properties of *variables*, *side-effects*, *values returned*, and *parameters*. The *variables* property describes the kinds of variables that are

	Function Class	
Property	Procedural	Pure Valued
variables	global, local	local
side-effects	yes	no
parameters	value, value-result, result	value
values returned	multiple	single

Table 2: A Taxonomy of Programming Language Functions

used by a function. The *side-effects* property is used to indicate whether the class of functions produces side-effects. The types of parameters and the number of values that are returned by a function are described by the *parameter* and *values returned* properties, respectively. *Pure Valued* functions are characterized by the fact that the variables used are local, the functions produce no side-effects, the parameters are value parameters and the functions return a single value. Note that a procedural function can effectively serve the role of a pure valued function if it can be ensured that the functions produce no side effects. This implies that the number of values must be singular. In this context, we assume that the modification of a value-result parameter or result parameter produces a side-effect.

A function in the C programming language has a signature (or prototype) of the form $\mathcal{R}$ f ($\mathcal{D}$) where $\mathcal{R}$ is the return type, and $\mathcal{D}$ is the input type of function f. For example, a function max could have a signature "`int max(int, int);`" Given a variable "x" of type $\mathcal{R}$, a parameter "a" of type $\mathcal{D}$, and an assignment operator "≅", a call to the function *f* has the form "**x ≅ f (a)**".

Let f be a pure valued function. The effect of calling the function is that a value is returned and assigned to the variable x. The corresponding *sp* semantics for the function call is

$$sp(\mathtt{x} := \mathtt{f(a)}, Q) = (\exists v :: Q^x_v \wedge x = \mathcal{A}(\mathtt{x} := \mathtt{f(a}^x_v\mathtt{)})). \quad (7)$$

This specification states that after the execution of an assignment statement using a function call, there exists some value v such that the textual substitution of every free occurrence of x with v in Q keeps Q true, and x takes the value of the evaluation $\mathcal{A}$ on $\mathtt{x} := \mathtt{f(a}^x_v\mathtt{)}$. Note that in the case where a pure valued function is called but not assigned that $sp(\mathtt{f(a)}, Q) = Q$.

4 Integrated Approach

Due to the mathematical nature of formal specification languages, formal methods have been described as time consuming and tedious. However, since the languages are well-defined, formal methods have been found to be amenable to automated processing. Semi formal methods are techniques for specifying system requirements and design using hierarchical decomposition. Most semi-formal methods have the property that the notations are graphical, facilitating ease of use in their application. The drawback of semi formal methods is that the notations are imprecise and ambiguous. This section describes an approach to reverse engineering that integrates the use of semi-formal methods and formal methods in order to utilize the complementary advantages of the notations.

4.1 Structured Analysis

Although the recent trend in software development has been to build systems using object-oriented technology, a majority of existing systems have been developed using imperative programming languages, such as C, FORTRAN, and COBOL. The procedural structure of these languages makes them amenable to the techniques offered by the *Structured Analysis and Design Technique* (SADT) [14]. In SADT, the focal point is the procedure or function. The analysis stage centers around high-level descriptions of the functionality of the system. During the design phase, the refinement and decomposition of the high-level description of functions yields more detailed descriptions of functions and procedures that incorporate implementation details. Finally, during the implementation phase, functions and procedures identified during design are decomposed into more specific functions.

When using SADT for reverse engineering activities, the structure of an implementation is abstracted into high-level graphical descriptions of functions known as call graphs or structure charts. These graphs depict the calling hierarchy of functions within a system. Further analysis of source involves analyzing the data that flows to and from various functions by constructing data flow diagrams. Our approach is to construct various graphical descriptions of a program, in most cases automatically, and then use those descriptions to guide the construction of formal specifications from the different parts identified by the graphical descriptions.

4.2 Applying formal techniques

The purpose of integrating the use of formal methods and semi-formal methods is two-fold. First, it is desirable to take advantage of the benefits of the complementary techniques. Second, by using a semi-formal technique to guide the formal technique, organization of the formal specifications will be based on the structure of an implementation. As such, in the case where formal specifications are warranted, the specifications can be directly associated with a graphical entity, while those parts of a module that do not require rigorous descriptions can be left unspecified (formally), with the descriptions of these modules being left to the semi-formalisms.

There are three guidelines that are followed when formally specifying a module. That is, the process of formally specifying a module consists of three steps or phases:

1. Local Analysis
2. Use Analysis
3. Global Analysis

During the *local analysis* phase, the calling hierarchy of a module is constructed and a skeletal formal specification is built, with the *sp* predicates left as parameterized transforms, that is, the transformations for *sp* are unevaluated. The objective is to gain a high-level understanding of the logical complexity of the given code. The second step, *use analysis*, is a recursive step where the three phases are applied to the functions and procedures *used* by the original module. This phase is characterized by the fact that the semantics of the *used* functions and procedures are determined before they are used by the original module. However, in many cases, where the semantics are either well-defined or the semantics are not critical, an unevaluated *sp* predicate can be used. For example, given a statement S and a precondition Q where the semantics of S are well-defined, instead of evaluating the transformation, we use $sp(S, Q)$ to represent the logical expression describing the semantics. In the *global analysis* phase, the *use analysis* information is combined with the *local analysis* information to obtain a global description of the original module. The global description, an expanded form of the skeleton formal specification constructed during the first phase, elaborates upon the semantics of a module by integrating the specifications constructed during the *use* analysis into the skeleton. This activity corresponds to removing the encapsulation provided by a procedure or function call.

Formal methods have been found to be amenable to automated processing. In addition, many techniques for abstracting semi-formal graphical specifications from code have been suggested [17]. In order to support our approach, we have been developing a system called AUTOSPEC. Currently, AUTOSPEC supports the construction of graphical specifications from C programs and is being extended to support the construction of formal specifications using the three step approach described above.

5 An Example

In this section we demonstrate the use of an integrated approach to modules from a mission control groulld-based system at the NASA Jet Propulsion Laboratory. The purpose of the code is to translate user commands into spacecraft mnemonics.

5.1 Local Analysis

Figure 4 gives the code for the translate procedure. An initial semiformal analysis of the translate code yields a calling graph as depicted in Figure 5, where the rectangles indicate functions, and the labels correspond to the function names given by the index to the right of the graph. From this initial analysis, we find that the translate function uses five functions including `initialize_interpreter`, `process_binary_output`, `inform_user`, `process_mnemonic_input`, `end_cmdxlt`, and `process_carg`. The translate function has four different modes: initialize, translate, control argument assignment, and error. For this analysis, we assume that we are only interested in the translate function in the translate mode. Thus, we are ignoring the initialization, control argument, and default modes in this analysis which correspond to the `INIT`, `CARG`, and default cases of the `switch` statement[1]. Therefore, we are left with specifying the `while` statement depicted in Figure 6, where labels have been added for convenience in the following discussion. Informally, the `translate` function in the translate mode is responsible for building a list of spacecraft instructions corresponding to interpreted commands by calling a function called `process_binary_output`.

An analysis of the code in Figure 6 using the sp rule for the `while` statement yields the following specification:

$$\neg(\texttt{args[0] != '0'}) \wedge (\exists i : 0 \le i : sp(S0^i, Q)), \tag{8}$$

where the expression (`args[0] != '0'`) has no side effects, and Q is the precondition to the statement S0. This specification states that after the `while` statement has been executed, the `args` array has a '0' as the first entry, and the statement S0 has been executed some number of iterations. Unfortulately, the specification in (8) is not very informative outside of identifying that the program uses an iterative construct. As such, an expansion of $sp(S0, Q)$ is warranted.

Using the labels shown in Figure 6, a specification of $sp(S0, Q)$ is given by

[1] Although in the context of this paper we have not defined the semantics of the `switch` statement, our investigations have included the construct.

```
struct msg *translate (int op, char *args)
{
    extern int dontoutput
    static struct project_parameters *pp;
    struct msg *mp = NULL;

    switch (op)
    {
        case INIT:              /* initialize the interpreter */
           pp = initialize_interpreter();
           break;

        case XLT:               /* interpret a message */
           while (args[0] != '\0')
           {
              if (process_mnemonic_input(&args, pp))
              {
                 if (mp == NULL)
                    mp = process_binary_output(pp);
                 else
                 {
                    mp->next = process_binary_output(pp);
                    mp = mp->next;
                 }
              }
              else
                 dontoutput = 1;
           }
           break;

        case CARG:              /*set a value for a control argument */
           process_carg(&args, pp);
           break;

        default:
           inform_user("internal error: bad op in translate");
           end_cmdxit(CMD_ERROR);
    }

    return(mp);
}
```

Figure 4: Translate Source Code

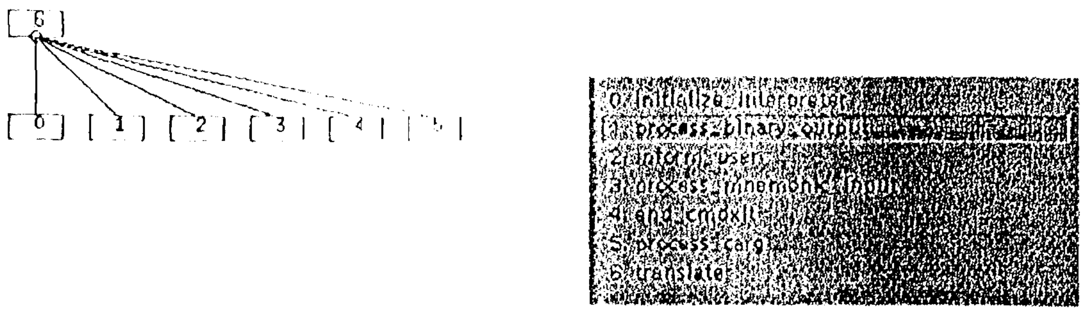


Figure 5: Translate

$$sp(\mathrm{S0}, Q) = sp(\mathrm{S1}, \mathcal{V}(B) \wedge sp(B, Q)) \vee sp(\mathrm{S2}, \neg\mathcal{V}(B) \wedge sp(B, Q)) \quad (9)$$

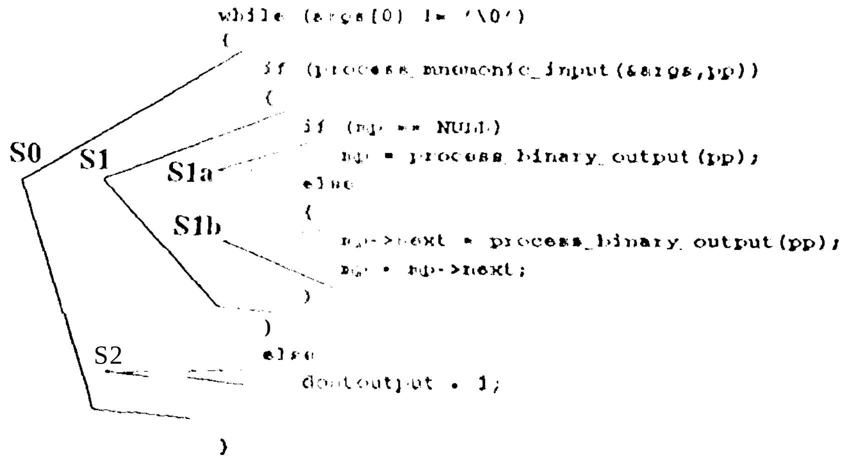
where B := `process_mnemonic_input(&args,pp)`. This specification states that after executing the statement **S0**, it will be true that either S1 was executed or **S2** was executed, where the semantics are determined by the preconditions $\mathcal{V}(B) \wedge sp(B, Q)$ and $\neg\mathcal{V}(B) \wedge sp(B, Q)$, respectively. So, in this case, either the **if** statement (S1) was executed or the assignment statement (**S2**) was executed. The specification makes explicit that the precondition $sp($`process_mnemonic_input(&args,pp)`$, Q)$ to the statement **S0** may contain a side effect. Note that if the function `process_mnemonic_input` has no side effect that

$$sp(\texttt{process\_mnemonic\_input(\&args,pp)}, Q) = Q.$$

Further expansion of $sp(\mathrm{S1}, \mathcal{V}(B) \wedge sp(B, Q))$, and $sp(S2, \neg\mathcal{V}(B) \wedge sp(B, Q))$ yield

$$\begin{aligned} sp(\mathrm{S1}, \mathcal{V}(B) \wedge sp(B, Q)) = \; & sp(\mathrm{S1a}, (mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B, Q)) \vee \\ & sp(\mathrm{S1b}, (mp \neq NULL) \wedge \mathcal{V}(B) \wedge sp(B, Q)), \end{aligned} \quad (10)$$

and



```
while (args[0] != '\0')
{
    if (process_mnemonic_input(&args,pp))
    {
        if (mp == NULL)
            mp = process_binary_output(pp);
        else
        {
            mp->next = process_binary_output(pp);
            mp = mp->next;
        }
    }
    else
        dontoutput = 1;
}
```

Figure 6: Translate Source Code

$$
\begin{aligned}
sp(\mathrm{S2}, \neg\mathcal{V}(B) \wedge sp(B,Q)) &= sp(\mathtt{dontoutput\ =\ 1}, \neg\mathcal{V}(B) \wedge sp(B,Q)) \\
&= (dontoutput = 1) \wedge (\neg\mathcal{V}(B) \wedge sp(B,Q))^{dontoutput}_{v}
\end{aligned}
\tag{11}
$$

respectively, where u is the value of dontoutput before executing S2. Equation (11) states that given that the expression '$\mathcal{V}(B) \wedge sp(B,Q)$' is true, either **S1a** has been executed or **S1b** has been executed, each depending on the added condition that either $(mp = NULL)$, or $(mp \neq NULL)$, respectively. On the other hand, Equation (12) states that given that the expression '$\neg\mathcal{V}(B) \wedge sp(B,Q)$' is true, execution of S2 results in the assignment of the variable '**dontoutput**' to be '1'.

The preliminary skeleton of the logical specification of the translation module can be constructed by substituting the Equations (11) and (12) back into the original Equation (10) such that

$$
\begin{aligned}
sp(\mathrm{S0},Q) = \; & sp(\mathrm{S1a}, (mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B,Q)) \vee \\
& sp(\mathrm{S1b}, \neg(mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B,Q)) \vee \\
& (dontoutput = 1) \wedge (\neg\mathcal{V}(B) \wedge sp(B,Q))^{dontoutput}_{v}
\end{aligned}
\tag{12}
$$

which states that in every iteration, one of three actions is executed, namely one of **S1a**, **S1b**, or

S2.

At this point in the analysis, since S1a and S1b are statements that depend on the specification of functions and procedures that are used by translate, it is appropriate to begin a *use analysis* for the translate function, where in this case, the function process_binary_output is analyzed.

In summary, during the local analysis phase for translate a graphical representation of the function was created with the intention of determining the calling hierarchy for the function. Next, a logical analysis was performed using top down approach that uses encapsulation with the intention of determining the logical complexity

5.2 Use Analysis

Use analysis involves the specification of functions that are used by a given object of study. In our example, given that the object of study is the translate function, use analysis involves specifying the functions used by translate. In this section we describe the function process_binary_output.

Figure 7 contains the source code for process_binary_output. The *use* analysis for this function involves three steps, each corresponding to the steps followed for translate. That is, we perform *local*, *use*, and *global* analyses on process_binary_output. The remainder of the process of analyzing process_binary_output is similar to the process used to analyze translate. However, in the interest of simplifying the analysis we shall ignore many of the details involved with analyzing process_binary_output and focus primarily the output characteristics. Note that the strict application of the rules for *sp* require a line by line construction of a specification. Here, we informally construct the specification with the understanding that all of the information can and should be constructed rigorously. Our main objective in this example analysis is to provide enough information about process_binary_output to be able to describe translate in a sufficient manner.

Consider the code of Figure 7 for process_binary_output. There are three statements that determine whether or not the output of the function is defined or not. These are indicated by the line numbers I, J, and K, respectively. Line I, for instance, has the interpretation that if space

```
    struct msg *process_binary_output (struct project_parameters *pp)
    {
        extern U16 *stem_entry;
        U16 code;
        U16 *ep;
        struct msg *mp

        Q = control_list,
        W = (U16 *)stack_base;
        S = (U32 *)mon_S;

        mp =(struct msg *)malloc(sizeof(struct msg) + MAX_MSG_BYTES);
I:      if (mp = NULL)
        {
            warn("process binary output: out of memory (mallocfailed)\n");
            end_cmdxlt( 1 ),
        }
        PUSHL(mp->msg_bits 1);   /* -1 for length field, written over later */

        ep = get_entry(get U32_Q());
        P = ep + 1;
        do
        {
            code = *P++,
            if ((code < 1) || (code > 32))
            {
                warn("bad code");
                end_cmdxlt(-1);
            }
            (*output_rtr[code])();
        } while (code != RFMS);
        mp->next = NULL;
        mp->msg_len = *(mp->msg_bits - 1);
J:      if (mp->msg_len > pp->max_msg_bits)
        {
            fail(TOO_MANY_BITS, NULL, NULL);
            free(mp);
            return(NULL);
        }
        mp->msg_num = 0;
        copy_space_filled("", mp->start, sizeof(mp->start));
        copy_space_filled("", mp->open, sizeof(mp->open));
        copy_space_filled("", mp->close, sizeof(mp->close));
        copy_space_filled(get_stem_and_title(stem_entry), mp->comment,
                          sizeof(mp->comment));
        mp->chksum = chksum(mp->msg_bits, FLD_LEN_OF(mp->msg_len)*2);
K:      return(mp);
    }
```

Figure'j: Process Binary Output Source

could not be allocated for the return object, the routine aborts, while line J forces the routine to return a NULL object due to some other error. Finally, the line K indicates a successful return of an object. Therefore, we can construct the following specification for process_binary_output:

$$\begin{aligned}
&sp(warn;\ end\_cmdxlt, (mp = NULL) \wedge Q) \vee \\
&sp(fail; free; return(NULL), (\texttt{mp->msg\_len > pp->max\_msg\_bits}) \wedge (mp \neq NULL) \wedge Q)) \vee \\
&sp(return(mp), (\texttt{mp->msg\_len < pp->max\_msg\_bits}) \wedge (mp \neq NULL) \wedge Q)
\end{aligned} \qquad (13)$$

which states that after executing process_binary_output either warn and end_cmdxlt were executed, the routine returned a NULL object, or the routine returned a valid object. Again, we stress that this specification is incomplete and only specifies a small slice of the functionality of the routine. Since this routine (along with trans_stat) are taken out of context, a full specification makes no contribution to this example.

5.3 Global Analysis

The final step in the analysis is to take the specification of Equation (13) and integrate it back into the skeleton specification of Equation (14). This specification is as follows

$$\begin{aligned}
sp(S0, Q) = {} & ((rep = NULL) \vee (mp = v)) \vee \\
& (((\texttt{mp->next} = NULL) \vee (\texttt{mp->next} = v)) \wedge (mp = \texttt{mp->next})) \vee \\
& (dontoutput = 1) \wedge (\neg B(B) \wedge sp(B, Q))_{v}^{dontoutput}
\end{aligned} \qquad (14)$$

where u is some new object. This specification states that after executing S0, the variable mp has either the value NULL or points to some new object, or mp->next has the value NULL or points to some new object with mp pointing to mp->next. Finally, if neither of those cases holds, it must be that dontoutput = 1. In the context of the specification of Equation (8), this specification means that after each iteration, a chain of messages is constructed or the dontoutput flag is set to 1. Note that in this specification we make the assumption that the pointer assignment behaves like a variable assignment. In this case there is no impact of making this assumption. However, there are semantics that are related specifically to pointers [?].

6 Related Work

Previously, formal approaches to reverse engineering have used the semantics of the weakest precondition predicate transformer *wp* as the underlying formalism of their technique. The *Maintainer's Assistant* uses a knowledge-based transformational approach to construct formal specifications from program code via the use of a Wide Spectrum Language (WSL) [6]. A WSL is a language that uses both specification and imperative language constructs. A knowledge base manages the correctness preserving transformations of concrete implementation constructs in a WSL to abstract specification constructs in the same WSL.

REDO [5] (Restructuring, Maintenance, Validation and Documentation of Software Systems) is an Espirit II project whose objective is to improve applications by making them more maintainable through the use of reverse engineering techniques. The approach used to reverse engineer COBOL, involves the development of general guidelines for the process of deriving objects and specifications from program code as well as providing a framework for formally reasoning about objects [18].

The "Loop ANalysis Tool for Recognizing Natural concepts" or LANTRN is an approach that uses a multi-step process to construct predicate logic annotations for loops. The analysis process involves the translation and normalization of loop programs into forms that are amenable to matching of various components of loops. A knowledge base or *plan library* is used to identify stereotypical loop *events*, where events come in the form of *basic events* and *augmentation events.*

The approach taken by the LANTRN system moves in the direction of making other plan-based approaches more formal in that, the final product of the loop analysis activity is the construction of a formal specification. The shortcomings of this approach are that the use of a knowledge-base requires constant updates to handle new cases, meaning that the size of the knowledge-base can become unmanageable. in addition, while the activity produces a formal specification, there is no formal basis for the verification that the specification of the plan matches the true semantics of a loop.

In the REDO and *Maintainer's Assistant* approaches, the applied formalisms are based on the semantics of the *weakest precondition* predicate transformer *wp* Some differences in applying *wp* and *sp* are that *wp* is a backward rule for program semantics and assumes a total correctness model of execution. However, the total correctness interpretation has no forward rule (i.e. no strongest *total postcondition stp* [7]). By using a partial correctness model of execution, both a forward rule (*sp*) and backward rule (*wlp*) can be used to verify and refine formal specifications generated by program understanding and reverse engineering tasks. The main difference between the two approaches is the ability to directly apply the strongest postcondition predicate transformer to code to construct formal specifications versus using the weakest precondition predicate transformer as a guideline for constructing formal specifications.

7 Conclusions and Future Investigations

Formal methods provide many benefits in the development of software. Automating the process of abstracting formal specifications from program code is sought but, unfortunately, not completely realizable as of yet. However, by providing the tools that support the reverse engineering of software, much can be learned about the functionality of a system.

Currently we are developing a system to support all of the techniques described in this paper called AUTOSPEC. In addition, we have been applying our techniques to a ground-based mission control system for controlling unmanned spacecraft at the NASA Jet Propulsion Laboratory. Our future investigations include the development of an approach to introducing abstraction into the specifications built using our reverse engineering technique.

References

[1] Wilma M. Osborne and Elliot J. Chikofsky. Fitting pieces to the maintenance puzzle. *IEEE Software*, 7(1):11-12, January 1990.

[2] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8-24, September 1990.

[3] B. H. C. Cheng. Applying formal methods in automated software engineering. *Journal of Computer and Software Engineering*, 2(2):137-164, 1994.

[4] Gerald C. Gannod and Betty H.C. Cheng. Facilitating the Maintenance of Safety-Critical Systems Using Formal Methods. *The International Journal of Software Engineering and Knowledge Engineering*, 4(2), 1994.

[5] K. Lano and P.T. Breuer. From Programs to Z Specifications. In John E. Nicholls, editor, *Z User Workshop*, pages 46–70. Springer-Verlag, 1989.

[6] M. Ward, F.W. Calliss, and M. Munro. The Maintainer's Assistant. In *Proceedings for the Conference on Software Maintenance*. IEEE, 1989.

[7] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[8] C. A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[9] Betty H.C. Cheng and Gerald C. Gannod. Abstraction of Formal Specifications from Program Code. In *Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*, pages 125–128. IEEE, 1991.

[10] Gerald C. Gannod and Betty H.C. Cheng. Strongest Postcondition as the Formal Basis for Reverse Engineering. In *Proceedings for the Second Working Conference on Reverse Engineering*, pages 188–197. IEEE, 1995.

[11] Gerald C. Gannod and Betty H.C. Cheng. Strongest Postcondition as the Formal Basis for Reverse Engineering. *(To appear in) Journal of Automated Software Engineering*, 1996.

[12] Eric Byrne. A Conceptual Foundation for Software Re-engineering. In *Proceedings for the Conference on Software Maintenance*, pages 226–235. IEEE, 1992.

[13] Eric J. Byrne and David A. Gustafson. A Software Re-engineering Process Model. In *COMPSAC*. ACM, 1992.

[14] E. Yourdon and L. Constantine. *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.

[15] Edsgar W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976

[16] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[17] Phillip Newcomb. Reengineering Procedural Into Data Flow Programs. In *Proceedings for the Second Working Conference on Reverse Engineering*, pages 32–38. IEEE, 1995.

[18] H.P. Haughton and K. Lano. Objects Revisited. In *Proceedings for the Conference on Software Maintenance*, pages 152–161. IEEE, 1991.